Australian National University

COMMUNICATING TASKS

Week 6 Laboratory for Concurrent and Distributed Systems

Uwe R. Zimmer based on material by Alistair Rendell

Pre-Laboratory Checklist

- ❑ **You have read this text before you come to your lab session.**
- ❑ **You understand and can utilize the live-span of tasks.**
- ❑ **You have a firm understanding of memory based synchronization.**

Objectives

This lab will introduce you to direct interaction (synchronous message passing) possibilities between tasks. For this we will also need to learn about who can communicate with whom and how long (or until when) each of those communication partners lives. For the keen students we will also look into options to "do a post-mortem" and to provide tasks "last wishes".

The key aspect of this lab is on message passing though. So make sure you fully understand how this works before you leave this lab.

Interlude: Communicating Tasks

In the second interlude for this lab we introduced the declaration of entries which will be used to communicate with such a task by synchronous message passing. Now we will introduce the receiving side of message passing to make it all work. Tasks are said to **accept** waiting calls to a message passing interface and when they do they are synchronized with the external caller (both tasks are said to be in a **rendezvous** block) and can directly exchange data. As both tasks need to be synchronized: whoever attempts to enter the rendezvous block first will need to wait for the communication partner to show up.

Syntactically tasks are declaring and accepting entry calls as such:

```
entry_declaration ::=
   [overriding_indicator] entry defining_identifier
         [(discrete_subtype_definition)] parameter_profile [aspect_specification];

accept_statement ::=
   accept entry_direct_name [(entry_index)] parameter_profile [do
      handled_sequence_of_statements
   end [entry_identifier]];

entry_index ::= expression
```

Let's have a look at the different options by looking at a complete program which uses synchronous message passing to hand over id's to tasks and to check that all tasks completed:

```
with Ada.Text_IO;  use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;

procedure Synchronize is

   Start_Up_Time : constant Time     := Clock;
   No_Of_Tasks   : constant Positive := 2;
   subtype Task_Range is Positive range 1 .. No_Of_Tasks;

   procedure Put_Line_w_Time (S : String) is

   begin
      Put_Line ("At" & Duration'Image (Clock - Start_Up_Time) & " s : " & S);
   end Put_Line_w_Time;

   procedure Put_Line_w_Time_n_Task_Id (Id : Task_Range; S : String) is

   begin
      Put_Line_w_Time ("Task" & Task_Range'Image (Id) & " " & S);
   end Put_Line_w_Time_n_Task_Id;

   task type Pinger is

      entry Hand_over_Id (Given_Id : Task_Range);
      entry Last_Sync;

   end Pinger;

   task body Pinger is

      Id : Task_Range;

   begin
      Put_Line_w_Time ("Task ? is waiting for a ""Hand_over_Id"" rendezvous");
      accept Hand_over_Id (Given_Id : Task_Range) do
         Id := Given_Id;
         Put_Line_w_Time_n_Task_Id (Id, "in a ""Hand_over_Id"" rendezvous");
      end Hand_over_Id;
      Put_Line_w_Time_n_Task_Id (Id, "after a ""Hand_over_Id"" rendezvous");

      for i in 1 .. 4 loop
         Put_Line_w_Time ("Task" & Positive'Image (Id) & " active.");
         delay Duration (Id);
      end loop;

      Put_Line_w_Time_n_Task_Id (Id, "waiting for a ""Last_Sync"" rendezvous");
      accept Last_Sync do
         Put_Line_w_Time_n_Task_Id (Id, "in a ""Last_Sync"" rendezvous");
      end Last_Sync;
      Put_Line_w_Time_n_Task_Id (Id, "after a ""Last_Sync"" rendezvous");

   end Pinger;

   Tasks : array (Task_Range) of Pinger;

begin
   for i in Tasks'Range loop
      Put_Line_w_Time_n_Task_Id (i, "is called at ""Hand_over_Id""");
      Tasks (i).Hand_over_Id (i);
      Put_Line_w_Time_n_Task_Id (i, "released main task from ""Hand_over_Id""");
   end loop;
   for i in reverse Tasks'Range loop
      Put_Line_w_Time_n_Task_Id (i, "is called at ""Last_Sync""");
      Tasks (i).Last_Sync;
      Put_Line_w_Time_n_Task_Id (i, "released main task from ""Last_Sync""");
   end loop;
end Synchronize;
```

Download and run this program. Study carefully the sequence of actions and make a note where and when tasks are waiting for communication partners.

Exercise 1: Message passing to synchronize critical regions

While message passing can be used to … well … pass messages, it can also be used to synchronize actions. In this case the message does not even need to have any contents (i.e. parameters). The following program illustrates this idea:

```
with Ada.Numerics.Discrete_Random; use Ada.Numerics;
with Ada.Text_IO;                  use Ada.Text_IO;

procedure Token_Ring is

   No_of_Nodes       : constant Positive := 8;
   Elements_per_Node : constant Positive := 100_000;

   type Element is new Long_Integer;

   Size : constant Positive := Elements_per_Node * No_of_Nodes;
   subtype Index is Positive range 1 .. Size;

   package Random_Elements is new Discrete_Random (Element);
   use Random_Elements;

   type Nodes is mod No_of_Nodes;

   task type Node is
      entry Handover_Id (Assigned_Id : Nodes);
      entry Token;
   end Node;

   Ring : array (Nodes) of Node;

   Random_Field : array (Index) of Element := (others => Element'Invalid_Value);
   Global_Max   : Element                  := Element'First;

   Element_Generator : Generator;

   function Partition_First (Id : Nodes) return Index is
                      (Index'First +  Natural (Id)      * Elements_per_Node);
   function Partition_Last  (Id : Nodes) return Index is
                      (Index'First + (Natural (Id) + 1) * Elements_per_Node - 1);
```

So far you have the infrastructure, created a ring of tasks and a global array of elements which can be accessed by any of those tasks. There is also a global variable `Global_Max` which gives you a hint of what this program is supposed to do.

Now to the interesting part of how those tasks communicate and synchronize:

```
   task body Node is

      Id : Nodes := Nodes'Invalid_Value;

   begin
      accept Handover_Id (Assigned_Id : Nodes) do
         Id := Assigned_Id;
      end Handover_Id;

      declare
         Next      : constant Nodes   := Id + 1;
         Local_Max :          Element := Element'First;

         subtype Partition is Index
                              range Partition_First (Id) .. Partition_Last (Id);
```

```
      begin
         accept Token;
         if Id /= Nodes'Last then
            Ring (Next).Token;
         end if;
         if Id = Nodes'Last then
            Ring (Next).Token;
         end if;
         accept Token;
         if Id /= Nodes'Last then
            Ring (Next).Token;
         end if;
      end;
   end Node;
begin
   Reset (Element_Generator);
   for e of Random_Field loop
      e := Random (Element_Generator);
   end loop;
   for n in Nodes loop
      Ring (n).Handover_Id (n);
   end loop;
   Ring (Ring'First).Token;
end Token_Ring;
```

The tasks in the ring are not doing anything particularly productive, but just seem to be busily sending messages around. Analyse what is going on here in terms of synchronization: Which task is active or waiting for messages at which times? Also important: will all tasks complete or could a task be left waiting for a message which never comes? If you can answer these question then you have understood the idea of the above program.

Now you can make use of your insights and make this program do something useful, like calculating a global maximum in a concurrent fashion. This is not much of a programming exercise, but focuses your attention on the synchronization features, hence the two code snippets which you will need to calculate the global maximum are provided below (these are commented-out code sections at the end of the source code which you will download).

```
      for e of Random_Field (Partition) loop
         Local_Max := Element'Max (Local_Max, e);
      end loop;
      Put_Line ("Task" & Nodes'Image (Id) & " reports a local maximum of:"
                                          & Element'Image (Local_Max));
```

and

```
      Global_Max := Element'Max (Global_Max, Local_Max);
      Put_Line ("Task" & Nodes'Image (Id) & " set a global maximum of:"
                                        & Element'Image (Global_Max));
```

Now place those code sections into the program where they will allow for maximal concurrency – without producing race conditions, i.e. potentially wrong results due to possible sequences of operations. What *exactly* can you state about the order of terminal outputs in your program? Most importantly: Which aspects of your program are reproducible and which aspects will be different every time you run a test?

Submit the archive `Token_Ring.zip` of the working version of your program to the *Submission-App* under "Lab 6 Token Ring" for a detailed code review by your peers and by us.

Exercise 2: Reduce

Reducing (or "folding") a list of values into a single value is a frequent operation. You learned how to do this iteratively – here in Ada:

```
generic
   type Element       is private;
   type Index         is (<>);
   type Element_Array is array (Index range <>) of Element;
   with function Combine (Left, Right : Element) return Element is <>;
function Reduce_Iterative (A : Element_Array) return Element;
```

and its implementation:

```
function Reduce_Iterative (A : Element_Array) return Element is
   Combination : Element := A (A'First);
begin
   for e of A (Index'Succ (A'First) .. A'Last) loop
      Combination := Combine (e, Combination);
   end loop;
   return Combination;
end Reduce_Iterative;
```

You also learnt how do this recursively - here in Ada (same specification, different name):

```
function Reduce_Recursive (A : Element_Array) return Element is
   (if    A'Length = 1 then A (A'First)
    elsif A'Length = 2 then Combine (A (A'First), A (A'Last))
                       else Combine (A (A'First),
                                     Reduce_Recursive
                                           (A (Index'Succ (A'First) .. A'Last))));
```

So far there is nothing new here (besides the syntax maybe). Yet this being a concurrency course, you can probably tell what is coming: How to implement this concurrently?

Some concurrent programming languages (e.g. Chapel) offer such concurrent reductions as language primitives, so:

$$\sum_{i=1}^{1,000,000} i^2$$

becomes:

```
+ reduce [i in 1..1000000] i ** 2.0
```

Ada does not provide concurrent reductions as a language primitive, so you need to help things a bit here.

Before you start programming, sit back for a moment and think whether it makes sense to perform such operations concurrently? How can you hope to gain performance? How much performance gain can you hope for in the best case?

If you don't see the answer straight away, then it might help to take out a piece of paper and draw a neat diagram. Check out your diagram: How many concurrent entities would you need? How many of them can work concurrently? How many sequential steps will the complete process require? If you are unsure about the answers you would not be able to implement it yet, so make sure to get those right first.

The specification of your concurrent reduce function looks just the same as with the previous two sequential versions:

```
generic
   type Element       is private;
   type Index         is (<>);
   type Element_Array is array (Index range <>) of Element;
   with function Combine (Left, Right : Element) return Element is <>;
function Reduce_Concurrent (A : Element_Array) return Element;
```

The implementation of it will be your job for this exercise.

Start by downloading the project `Reduce` from the web-site. It will come with a handy test framework to make sure that your results are matching those of the sequential version.

If you are stuck or need some more inspiration have a look at this task definition:

```
task type Combiner (Slice_Begin, Slice_End : Index) is
   entry Combination (E : out Element);
end Combiner;
task body Combiner is
   Result : constant Element := Reduce_Concurrent (A (Slice_Begin .. Slice_End));
begin
   accept Combination (E : out Element) do
      E := Result;
   end Combination;
end Combiner;
```

This should get you up to speed and you only need to puzzle the parts together now in a meaningful concurrent way.

There is one little type-specific trick which you might need, as the Index type is not necessarily numeric given the above polymorphic definition. Hence you need to translate the index value into a discrete numeric value first in order to calculate where the middle of a your array is, i.e.

```
Middle : constant Index := Index'Val (Index'Pos (A'First) + A'Length / 2);
```

which means your array `A` could be divided into two slices from `A'First` to `Index'Pred (Middle)` and then from `Middle` to `A'Last`.

For the advanced and interested: the position value of a discrete item will relate to the position of this item as defined for instance in a sequence of enumerated values, meaning it will not refer to the representation value of such an item in memory – those two aspects of a type need to be strictly separated (which is important if you program "close to hardware"). You can for instance have an enumeration of colours where their position values are their places inside this enumeration, yet their representational values could be the colour-channel values which define those colours on a screen.

Submit the archive `Reduce.zip` of the working version of your program to the *SubmissionApp* under "Lab 6 Reduce" for a detailed code review by your peers and by us.

Exercise 3: Task finalizations

You probably wondered where we will de-allocate those pointers which we allocated to create dynamic tasks in the previous lab. Multiple ways of doing this:

- You don't and re-use those tasks for the next job. You need to make sure that your pointers are not running out of scope, which means those tasks need to be declared top-level.
- You make sure that the task terminated (i.e. by making it synchronize to something at the end), and then de-allocate the pointer manually.
- You leave it hanging and hope for a memory garbage collector to come around eventually. This could take some time though, as – while optional garbage collection is an option for the compiler providers (one of the very few choices for compilers in Ada) – nobody ever implemented one, as there is no demand for it in the high-integrity and real-time programming community where Ada is most dominantly used.
- You program your own, automated de-allocation method. Ada provides all the "hooks" to come up with your own implementation and for inspiration you could read up on current languages memory management discussions like for instance in the recent programming languages Rust or Swift. This gives you all options.

This exercise is about using the task finalization methods which are automatically called when a task completed and is to be finalized. Let's first have a look how the finalization method can be invoked for your specific tasks:

```
with Ada.Task_Identification; use Ada.Task_Identification;

package Scoped_Tasks_Finalizer is

   protected Finalizer is
      procedure Register (Id : Task_Id);
   end Finalizer;

end Scoped_Tasks_Finalizer;
```

and

```
with Ada.Exceptions;       use Ada.Exceptions;
with Ada.Task_Termination; use Ada.Task_Termination;
with Ada.Text_IO;          use Ada.Text_IO;

package body Scoped_Tasks_Finalizer is

   protected body Finalizer is

      procedure Last_Wish (Cause : Cause_Of_Termination;
                           Id    : Task_Id;
                           X     : Exception_Occurrence) is
      begin
         case Cause is
            when Normal              => Put_Line
                                 ("Task " & Image (Id) & " came to a peaceful end.");
            when Abnormal            => Put_Line
                           ("Somethings really nasty happend to task " & Image (Id));
            when Unhandled_Exception => Put_Line
         ("Unhandled exception " & Exception_Name (X) & " in task " & Image (Id));
         end case;
      end Last_Wish;

      procedure Register (Id : Task_Id) is

      begin
         Set_Specific_Handler (T => Id, Handler => Last_Wish'Access);
      end Register;
   end Finalizer;
end Scoped_Tasks_Finalizer;
```

The routine `Set_Specific_Handler` registers your own (protected) procedure to be called when a specific task terminates. As this package makes no assumption about the task itself, you can use this package for any task in your code. The most common purpose of such a structure is to provide an additional safety net for tasks which are not terminating gracefully or to clean up any resources which are associated with this task (and are for some reason not easily manageable from within the task) … like for instance the pointer variable to the task itself.

For this purpose we could for instance take note of all the pointer variables which we allocated and keep them in a hashed map structure (remember your earlier courses about heaps and maps?). We then use the task id as a key to look up the pointer which we need to de-allocate. This then looks a little more involved:

```
with Ada.Containers.Hashed_Maps; use Ada.Containers;
with Ada.Strings.Hash;           use Ada.Strings;
with Ada.Task_Identification;    use Ada.Task_Identification;
with Dynamic_Tasks;              use Dynamic_Tasks;

package Dynamic_Tasks_Finalizer is

   function Task_Id_Hash (Key : Task_Id) return Hash_Type is (Hash (Image (Key)));

   package Task_Ptr_Map is new Hashed_Maps (Key_Type        => Task_Id,
                                           Element_Type    => Dynamic_Task_Ptr,
                                           Hash            => Task_Id_Hash,
                                           Equivalent_Keys => "=");

   protected Finalizer_Deallocator is

      procedure Register (Ptr : Dynamic_Task_Ptr);

   private
      Running_Tasks : Task_Ptr_Map.Map;
   end Finalizer_Deallocator;

end Dynamic_Tasks_Finalizer;
```

and

```
with Ada.Exceptions;             use Ada.Exceptions;
with Ada.Task_Termination;       use Ada.Task_Termination;
with Ada.Text_IO;                use Ada.Text_IO;
with Ada.Unchecked_Deallocation; use Ada;

package body Dynamic_Tasks_Finalizer is

   procedure Free_Task_w_Discriminant is new Unchecked_Deallocation
                                            (Dynamic_Task, Dynamic_Task_Ptr);

   protected body Finalizer_Deallocator is

      procedure Last_Wish (Cause : Cause_Of_Termination;
                           Id    : Task_Id;
                           X     : Exception_Occurrence) is

         Ptr : Dynamic_Task_Ptr := Running_Tasks.Element (Key => Id);

      begin
         Running_Tasks.Delete (Key => Id);
         Free_Task_w_Discriminant (Ptr);
         case Cause is
            when Normal              => Put_Line
                                 ("Task " & Image (Id) & " came to a peaceful end.");
            when Abnormal            => Put_Line
                           ("Somethings really nasty happend to task " & Image (Id));
            when Unhandled_Exception => Put_Line
         ("Unhandled exception " & Exception_Name (X) & " in task " & Image (Id));
         end case;
         Put_Line ("Cleaned up dynamic task " & Image (Id));
      end Last_Wish;
```

```
      procedure Register (Ptr : Dynamic_Task_Ptr) is
         Task_Key : constant Task_Id := Ptr'Identity;
      begin
         Running_Tasks.Insert (Key => Task_Key, New_Item => Ptr);
         Set_Specific_Handler (T   => Task_Key, Handler  => Last_Wish'Access);
      end Register;
   end Finalizer_Deallocator;
end Dynamic_Tasks_Finalizer;
```

If you further assume that you have some tasks which are designed to be in scope and some tasks which are designed to be dynamically allocated:

```
package Scoped_Tasks is
   type    Colours     is (Red, Green, Blue, Pink);
   subtype Few_Colours is Colours range Red .. Blue;
   task type Scoped_Task is
      entry Hand_over_Task_Id (Set_Id : Colours);
   end Scoped_Task;
   Scoped_Task_is_not_too_well : exception;
end Scoped_Tasks;
```

and

```
package Dynamic_Tasks is
   subtype Dynamic_Id_Range is Positive         range 1 .. 1_000;
   subtype Few_Ids          is Dynamic_Id_Range range 1 .. 3;
   task type Dynamic_Task (Id : Dynamic_Id_Range) is
      entry Start;
   end Dynamic_Task;
   type Dynamic_Task_Ptr is access Dynamic_Task;
   Dynamic_Task_is_not_too_well : exception;
end Dynamic_Tasks;
```

you can then connect those tasks with your preferred finalization method:

```
with Dynamic_Tasks;           use Dynamic_Tasks;
with Dynamic_Tasks_Finalizer; use Dynamic_Tasks_Finalizer;
with Scoped_Tasks;            use Scoped_Tasks;
with Scoped_Tasks_Finalizer;  use Scoped_Tasks_Finalizer;
procedure Scoped_vs_Dynamic_Finalization is
   Array_of_Scoped_Tasks  : array (Few_Colours) of Scoped_Task;
   Array_of_Dynamic_Tasks : array (Few_Ids)     of Dynamic_Task_Ptr;
begin
   for i in Array_of_Scoped_Tasks'Range loop
      Finalizer.Register (Id => Array_of_Scoped_Tasks (i)'Identity);
      Array_of_Scoped_Tasks (i).Hand_over_Task_Id (i);
   end loop;
   for i in Array_of_Dynamic_Tasks'Range loop
      Array_of_Dynamic_Tasks (i).all.Start;
      Finalizer_Deallocator.Register (Array_of_Dynamic_Tasks (i));
      Array_of_Dynamic_Tasks (i) := new Dynamic_Task (i);
   end loop;
end Scoped_vs_Dynamic_Finalization;
```

Unfortunately, somebody messed up the order in which tasks are allocated, registered and started. So if you run the above program, you will likely see some form of memory violation. Repair it and then run the whole program.

You will also have noticed that the finalization routines can detect whether a tasks terminated due to an uncaught exception. Provoke this behaviour by removing some of the exception handlers from the code. Is this a reasonable way to react to exceptions? Explain to your tutor what you think and when (or if) you would use it.

Obviously the effort to handle dynamic tasks is much higher than handling scoped tasks. So if you have a choice you will always use the much safer and cleaner option of scoped tasks. Yet there must be some use for those dynamic tasks (otherwise the people who designed the language probably wouldn't have introduced them?). Identify a use case for it and explain this one to your tutor. Hint: a variable number of task instances at run-time is not the answer – you can easily create those with declared, scoped tasks.

MAKE SURE YOU LOGOUT TO TERMINATE YOUR SESSION!

Outlook

Next week you will make your tasks communicate by sharing data in the context of dynamic servers